DOCUMENT RESUME

ED 183 202 IR 008 146

AUTHOR Cox, John P.
TITLE Common PILOT Manual. The Illinois Series on Educational Applications of Computers, Number 34.
INSTITUTION Illinois Univ., Urbana. Dept. of Secondary Education.
PUB DATE Jul 79
NOTE 36p.

EDRS PRICE MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS *Computer Assisted Instruction: Computer Programs: Instructional Design: *Manuals: *Material Development: Programing: *Programing Languages: *Teacher Develcped Materials
IDENTIFIERS Computer Software: Microcomputers

ABSTRACT

This manual presents an overview of the PILOT educational computer language, a simplified teacher-directed alternative to the use of BASIC for designing computer assisted instruction, and provides examples to illustrate its features. These features are described in terms of format, labels, numeric variables, alphanumeric variables, system variables, expressions and operators, functions, op-codes or commands, entering PILOT programs on the Horizon computer system, and running PILOT programs. Appendices describe proper use of vectors or arrays in PILOT programs, and exponential notation and ASCII code. (MV)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Number 34

The Illinois Series on Educational Applications of Computers

COMMON PILOT MANUAL

John P. Cox

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY
j. Richard Dennis
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

University of Illinois
at Urbana-Champaign
Department of Secondary Education
Urbana, Illinois

July, 1979

ED183202

IR008146

TABLE OF CONTENTS

Topic	Page
Format	1
Labels	2
Numeric Variables	3
Alphanumeric Variables	4
System Variables	5
Expressions and Operators	6
Functions	8
Op-Codes	11
Problem Command (PR)	11
Remark Command (R)	11
Dimension Command (D)	12
Match Command (M)	12
Compute Command (C)	16
Type Command (T)	17
Jump Command (J)	21
Accept Command (A)	22
Use Command (U)	23
End Command (E)	24
Execute Indirect Command (XI)	24
GOTO Command	26
Escape or @ Command	26
Entering Pilot Programs on the Horizon	28
Running Pilot Programs	30
Appendices	31

ISEAC Number 34

COMMON PILOT MANUAL

John P. Cox

The Pilot educational computer language represents a possible alternative to the use of BASIC in computer-assisted instruction lessons. Designed for teachers to use for CAI, this language could become a viable first exposure to educational computing which is much easier for teachers to understand and to use than BASIC for educational applications. The PILOT language was created at the San Francisco Medical Center; the version described herein was created at Western Washington University after consultation with a number of serious users of CAI. The current document describes PILOT as implemented on the Horizon system; completely compatible versions of this PILOT are available on a number of other commonly available micro and minicomputers. This manual presents an overview of the language with examples to help illustrate its features. The interested programmer should also refer to the COMMON PILOT REFERENCE MANUAL (available from Western Washington University, Bellingham, Wa. 98225). Some of the material in this publication is copyright by Western Washington University.

Format

There are only 11 commands (op-codes) which are used in the Pilot language. Each is discussed in the sections below. However, the successful use of the language demands that the programmer become familiar with the ways of modifying and/or conditioning these instructions to allow more versatility.

Normal execution of a Pilot program begins with the first instruction and continues sequentially. Several instructions can cause jumps to another part of the program, in some cases the student can cause such jumps, and individual instructions can be skipped if the criteria established by conditioners on that line are not met.

Each instruction is a single line of 2-72 characters. Op-code and other parts of the line must be capital letters, but text can be in lower case. (For example, if a teacher wanted to let a student know an answer was correct, the command:

```
T:  Your answer is right.
```

would cause the message "Your answer is right." to be printed on the system terminal.)

Labels

In order to identify certain instructions (so they can be referenced by the student or a jump command) a label can be included as part of an instruction line. When used, the label must be preceded by an asterisk and appear at the beginning of a line. The label itself can be 1-6 characters in length, the first of which must be a letter. The other characters may be letters of numbers (no spaces or punctuation) and all letters must be capitalized. The label can be placed on a line by itself (preceded by an asterisk) or can precede the op-code on a line of instruction. When the latter method is used, the label should be followed by a space to separate it from the op-code. Examples:

```
*SECT1
T:  Good Morning.
*SECT2 T:  Good Afternoon.
```

When a label is referenced by a jump instruction, the asterisk is not included (it is not part of the label name, but serves to tell the computer that a label follows). To cause the computer to jump to the above set of instructions, the programmer could use:

```
J:  SECT1
```

which would cause a break in the normal sequential execution of instructions and the computer would resume execution at the line labelled SECT1.

Numeric Variables

Variables can be named using a single, capitalized letter or using a capitalized letter immediately followed by a single digit (A, C1, DØ, etc.). Each variable must be assigned a value before attempting to use it elsewhere in the program (this is accomplished using the compute instruction,

```
C:  C1 = 5
```

and the variable name must be used on the left-hand side before it can be referenced on the right-hand side or in another instruction.)

Sometimes it is convenient to set up an array of variable locations identified by a variable name and subscripts. Pilot allows the use of one or two dimensional arrays (1 or 2 subscripts). These arrays are established by using the dimension command (for example:

```
D:  X1(20,20)
```

sets up a two dimensional array named X1 with locations named X1(0,0), X1(0,1), X1(0,2) . . . X1(0,20), X1(1,0), X1(1,1) . . . X1(1,20), X1(2,0) . . . X1(20,20) all of which can be used to store numeric values.)

The lowest subscript for an array is always 0 and the largest allowed subscript is established by the dimension command (with a maximum value of 999). Thus array X1 above has 441 subscripted locations. The subscripts can be constants or variables in referencing one of the array locations, but not expressions. Array variables cannot be used in a type or an accept command. (See Appendix A for further discussion of array usage.)

Alphanumeric Variables

Alphanumeric variables (sometimes called strings) are used to store a group of characters rather than a group of numbers. The variable name consists of one capital letter followed by a dollar sign or a capital letter followed by a digit and a dollar sign (A$, B1$, etc.). Every alphanumeric variable must be dimensioned before it is used. The dimension statement establishes the maximum allowable length for the variable. No variable can hold more than 999 characters. For example:

```
D:  C$(100)
```

establishes the alphanumeric variable C$ which has a maximum length of 100 characters. If an attempt is made to place more than the allowed characters into an alphanumeric variable, the computer will discard the extra characters (truncate) on the right. For example:

```
D:  X$(3)          (sets up length of three)

C:  X$="ABCDE"     (attempts to place 5 characters in the
                    variable)
```

will result in the value "ABC" placed into X$ since only 3 characters are permitted. When assigning a value to an alphanumeric variable or comparing its value to a group of characters, always enclose the characters in quotation marks as shown above. (This means that quotation marks cannot appear within the group of characters. However, this can be overcome--see the section below on the CHR function.)

Sometimes it is necessary to examine or change part of an alphanumeric variable's characters without disturbing the rest. This can be accomplished by using subscripts after the variable name. The first subscript defines the starting position of the group of characters being referenced. The second subscript states how many characters are to be examined or changed.

(If left out, the computer assumes a value of 1 for the second subscript.) Subscripts can be numeric constants or variables, but not expressions. They can range from 1-999, but should not exceed the maximum length set up for the referenced alphanumeric variable. For example:

```
D:  X$(6)          (sets up maximum length of 6)
D:  F$(3)          ( "   "     "      "    "  3)
C:  X$="ABCDEF"    (defines X$)
C:  F$=X$(2,3)     (places the value "BCD" into F$)
```

If, instead of the last line, we used either C: F$=X$(5,1) or C: F$= X$(5) then the value "E" would be placed in F$.

When changing the values stored in an alphanumeric variable, the subscripts can be used on the left-hand side of the equation. If too many characters are used, those on the right are ignored. If too few characters are used, spaces are added to complete the length. For example:

```
D:  X$(5)          (sets up maximum length of 5)
D:  F$(3)          ( "   "     "      "    "  3)
C:  X$="ABCDE"     (defines variable)
C:  F$="XYZ"       (   "       "    )
C:  X$(2,3)=F$     (changes value in X$ to "AXYZE")
```

If, instead of the last line, we used C: X$(2,2)=F$ then the value "AXYDE" would be placed in X$. If, instead, we used C: X$(1,4)=F$ then the value "XYZ E" would be placed in X$.

System Variables

Every time a student types in an answer, two system variables are updated. These variables can be referenced in the program and can be useful to the programmer.

%A is used as a counter and contains a numeric value of 0-9. Each time an accept instruction (used to input student responses) is

executed, this counter is incremented or reset to 1. If the same instruction is executed again before another accept statement is executed, the counter is incremented to 2. This counter has a maximum value of 356, but the value of 9 is the maximum for checks. The counter value can be reset by the programmer (for example C: %A=0 would set the value back to 0) and can be referenced in other instructions to determine if some action needs to be taken to help the student. For example:

T(%A > 3): The answer should be 4*5=20.

would cause the correct answer to be printed on the terminal only after the student had unsuccessfully tried to answer the question four times.

%B is used to store the actual answer which a student types in and is an alphanumeric variable. Some editing usually occurs to the student's response before it is placed in %B (see the section below on the accept instruction for the exception). Normally, any spaces preceding the first character or following the last character are removed and, if the student put more than one space between characters, these spaces are compressed into one space. Other editing options can be set by the program instruction (see the section below for a discussion of these options). Use of this system variable can reduce the number of alphanumeric variables used by a Pilot program (only answers which are needed later in the program need be stored).

Expressions and Operators

An expression is any combination of variables, numbers, and legitimate operators. For example: A+B-7 or C/3+D-7.23+2*(X-8). Operators which may be used with numeric variables are:

+ addition

- subtraction or negation A-B or -C+5

* multiplication

/ division

** exponent; X**3 means X to the third power;
fractional and negative exponents are allowed:
Y**2.6 or Z**(1/5)

Operators which may be used with string or alphanumeric variables: !! concatenation; joins the second group of characters to the right of the first group; for example:

```
D:  S$(5)          (sets length of 5)
D:  F$(3)          ( "     "    " 3)
D:  X$(1)          ( "     "    " 1)
C:  S$=""          (assigns string as empty--no
                   characters)
C:  F$="XYZ"       (assigns value of "XYZ")
C:  X$="A"         (   "      "   "  "A")
C:  S$=X$!!F$
```

will cause S$ to be assigned a value of "AXYZ." If we replace the last line with C: S$=X$!!"BC"!!F$(2,2) then the value of S$ will be "ABCYZ "

When logical or relational operators are used to combine such expressions, the resulting statement is called a relational expression. When relational operators are used to compare two quantities, both quantities must be numeric or both must be alphanumeric (you cannot compare X with S$). The relational operators are:

= equals

< less than

> greater than

<> not equal

<= less than or equal

>= greater than or equal

The result of the comparison is assigned a value of 1 if true and a value of 0 if false.

Logical operators can be used to combine more than one relational expression and make the result dependent on the truth of each. The logical operators are:

~ negation--changes the truth value of the relation it preceeds; if A < B is true, then ~(A < B) is false.

! or--the statement is true unless both relations are false.

& and--the statement is true only when both relations are true.

Operators are evaluated from left to right in any statement (except for exponentiation which goes from right to left), inner parentheses are evaluated first, and operators are evaluated at each step in the following order:

~

**

*/

+-!!

=, > , < , <> , <=, >=

!&

If complicated statements must be used, it is wise to test the version you devise by using some numbers which you can double-check before assuming that the computer will evaluate the statement the way you think it will!

Functions

Pilot also allows the use of 17 built-in functions which can be included in expressions. These functions will carry out numerical calculation or string manipulations automatically. The functions are:

ABS(X) returns the absolute value of the variable X; this value is also a number, but is always > = 0

ASC(S$) returns the ASCII code value of the character in S$; this value is an integer between 0 and 255, inclusive (see the table in Appendix B); if S$ has more than one character stored in it, only the first character is considered

ATN(X) returns the arctangent of X; this value is the number of radians which has a tangent of X

CHR(X) returns the character which has an ASCII code value of X; this can be used to load such special characters as backspace, line feed, quotation marks, etc. into an alphanumeric variable--

```
D:  S$(1)        (sets up maximum characters
                  allowed in S$)
C:  S$=CHR(13)   (loads a carriage return in-
                  to S$)
```

Other special characters of interest are 12--form feed or clear screen, 8--backspace, 34--quotation marks, 10--line feed. These will vary with the terminal used.

X must be an integer between 0 and 255, inclusive.

EXP(X) returns the value of e**X; this value is always a positive number

FIX(X) returns the value of X, truncated to reflect only the integer value; FIX(3.8) would return a value of 3; FIX(-5.6) would return a value of -5

FLO(S$) returns the numeric value of the first numerals encountered in the characters stored in S$; this value depends on the way in which the string is stored; if S$ has a value of "14 years and two months," then FLO(S$)=14; if S$ has a value of "Fourteen years and 2 months" then FLO(S$)=2; if S$ has a value of "5.92E+02" then FLO(S$)=5.92*10**2 (see Appendix B for a discussion of exponential notation)

INS(X,S$,S2$) returns the starting position of the substring, S2$ within the group of characters in S$, with the search starting from the Xth character in S$; this value is an integer and 0 is returned if the substring does not appear in S$; the variables used for S$ and S2$ must be unsubscripted (the whole string must be used) and the current length of each variable must be 255 or less; for example:

```
D:  S$(5)              (assigns maximum length
                       to S$)
D:  S2$(2)             (assigns maximum length
                       to S2$)
C:  S$="ABCDE"         (assigns value to S$)
C:  S2$="CD"           (   "      "   "  S2$)
C:  X=1                (   "      "   "  X)
C:  A=INS(X,S$,S2$)    (returns value of 3 for A)
```

INT(X) returns the integer which is less than or equal to X; for example, INT(-3.3)=-4, INT(5.7)=5

LEN(S$) returns the current length of S$; this value is a number, and always is positive and an integer or 0

LOG(X) returns the base 10 logarithm of X

LN(X) returns the logarithm of X in base e

RND(X) selects a random number between 0 and 1; X should also be less than 1 and greater than or equal to 0

SGN(X) returns a number which indicates the sign of X; this number is 1 if X is positive, -1 if X is negative and 0 when X=0

SIN(X) returns the sine of X; this is always a value between 1 and -1 inclusive; X is a measure of the angle in radians

SQR(X) returns the square root of X; this number is always the positive square root; X should be greater than or equal to 0

STR(X) returns the string or character representation for the value of X; this representation is always a group of characters, not a numerical value for computation; only one STR usage may appear in any expression; for example, if X = 135 then STR (X) returns the characters "135"

The programmer should be careful to use only numbers or numeric variables as function arguments for the value of X and only strings of characters or alphanumeric variables for S$ and S2$. Remember that it is not permissible to mix variable types in an expression, so simple uses of the functions are better than complex. Breaking complex expressions down into more simple ones is also good practice since it makes the algorithm used to do computations easier to understand.

Problem Command (PR)

The op-code PR: is used to mark the beginning of a new problem in an instructional program, to establish edit options for student input, and to allow changes in the normal sequential execution of programs. The options which are stated on a PR: statement remain in effect until the next PR: statement is executed. If no options are placed on a PR: statement, no change occurs. If options are declared on a PR: statement, then those options become the only options in effect.

PR: U	(causes all letters input by the student to be converted to upper case)
PR: L	(causes all letters input by the student to be converted to lower case)
PR: S	(causes all spaces input by the student to be removed)
PR: G	(permits the student to enter the GOTO Command--see the section below)
PR: W	(erases all the labels which have been used earlier in the program--allows the programmer to use the same labels more than once in a program and saves memory space)
PR: E	(permits the student to use the escape command (@)--see the section below)
PR:	(leaves the options previously declared in effect; at the beginning of a program, no options would be in effect)
PR: U S G	(combinations of options are allowed--this would cause all three options to go into effect)

Other option combinations are possible. The PR: statement can serve as the destination for the jump command. The command J: @P causes the computer to skip execution of all instructions until it finds the next PR: statement.

Remark Command (R)

The op code R: is used to document programs by allowing the programmer to insert remarks so other people can understand how the program works.

These remarks are not executed, but if the program is listed, they are present in the program to explain the algorithms used and the purpose of variables. For example:

```
R:  The variable X represents a random number.
C:  X=RND(.25)
```

Dimension Command (D)

The op code D: is used to dimension (reserve memory space) for string or alphanumeric variables and for numeric arrays. Only one variable may be placed on a dimension statement. The length of alphanumeric variables must be between 1 and 999, inclusive. Arrays may be one or two dimensional (one or two subscripts). Each subscript must be between 1 and 999 inclusive. Examples:

```
D:  S$(256)     (reserves 256 memory locations for
                characters in S$)
D:  X(10)       (reserves locations for X(0), X(1),
                X(2), . . . X(10) each of which is
                a numeric variable)
D:  X1(2,2)     (reserves locations for X1(0,0),
                X1(0,1), X1(0,2), X1(1,0), X1(1,1),
                X1(1,2), X1(2,0), X1(2,1), and
                X1(2,2) each of which is a numeric
                variable)
```

These alphanumeric variables and arrays must appear in a dimension statement before being used elsewhere in the program. It is good practice to place all of them at the beginning of a program along with remarks to describe how they are used. If they are executed more than once during a program, much memory space will be wasted.

Match Command (M)

The match op-code M: is used to compare the student's response with an expected correct answer. Other statements can be conditioned so as to execute only when the match is successful or only when the match is unsuccessful.

Numerical matches may be done using relational expressions to condition other instructions. For example:

```
C:  X=0          (sets initial value for numeric variable
                 X)
A:  #X           (inputs student answer and places the
                 first number into the variable X--see
                 the section below on the Accept command)
T(X=3.14):  Yes, that is a reasonable value for pi.
                 (causes the message to be output to the
                 student only if the student typed in
                 3.14 as part of the answer)
```

The main purpose of the match statement is in checking answers which are not numeric. The M: statement can be used to check for the occurrence of a word or words within the student's answer and can be used to allow for simple types of spelling mistakes in entering these words.

```
M:  HELLO        (matches successfully with the student
                 answer if the sequence of letters "HELLO"
                 appears anywhere in the answer; if the
                 student entered HELLO; HELLO, COMPUTER;
                 or I SAY HELLO, the match would be suc-
                 cessful; however, it will also match
                 with OTHELLO)
```

The programmer should be sure that the match statements agree with the PR: options in effect. In the above example, if the L option had been declared on the previous PR: statement, then the student's answer would be converted to lower case and would not match correctly. Lower case can be used on the M: statement in this case.

```
M:  %NO%         (the percent sign indicates that the
                 letters NO must be preceded and fol-
                 lowed by a space or they must be the
                 first word or last word in the student
                 input or the only word in the student
                 input; this would match successfully
                 with THERE IS NO ANSWER or NO THEY
                 AREN'T.  However, if the S option was
                 declared on the previous PR: statement,
                 all spaces are removed so this would
                 match only with the word NO)
```

M: %NO (matches the student answer only if the letters NO are the first two letters of the response or are preceded by a space; this would match with IT IS NOT. If the S option is not in effect; it would match with NOR, but not with ABNORMAL)

M: NO% (matches with the student answer only if the letters NO are the last two characters of the response or are followed by a space; this would match NO IT ISN'T if the S option is not in effect; it would match THE ANSWER IS NO and I LIVE IN RENO but would not match NONE)

Sometimes spelling errors are to be overlooked in judging student answers. This can be accomplished by inserting an * for letters which the programmer does not care about. For example:

M: H*RE (matches the student answer if the response includes a group of four characters which contain H followed by another character followed by RE--HIRE, HARE, SHORE would all be possible matches)

M: OX***N (matches any group of six characters with OX as the first two characters of the group followed by three other characters and the letter N)

The use of the spelling modifier on the Match Command allows for two common spelling errors: one incorrect letter in the word or two letters which are interchanged. For example:

MS: HIRE (matches with any group of four letters containing at least three of the letters HIRE in their proper positions such as SIRE, SHARE, HIPES, HIROSHIMA; also matches any group of four letters containing all four letters HIRE, but with two of the letters interchanged such as HIER, CHERI, and INHERIT)

Sometimes the correct answer to a question will contain more than one word. Pilot allows the programmer to combine words in several ways.

M: RED&BLUE	(matches only if the letters RED appear in the student's response somewhere before the letters BLUE; both words must appear in that order)
M: RED MY: BLUE	(using these two statements slightly modifies the situation from above; if the letters RED appear anywhere in the student's response, then the computer will check to see if the letters BLUE appear anywhere; this combination results in a correct match only if both words appear, but the order now is not considered)
M: HILLS!VALLEYS	(matches only if either the letters HILLS appear in the student's response or the letters VALLEYS appear or both words appear in any order; the correct match occurs if either or both words appear anywhere in the student's response)

More than two words can appear on such M: statements, but be careful when combining & and ! on the same line. The conditions are checked two at a time starting from the left.

M: HILLS&VALLEYS!MOUNTAINS	(matches if the words HILLS and VALLEYS appear in that order somewhere in the student's response or if the word MOUNTAINS appears or if both conditions are met)

The match statement can also be used as a destination for the jump statement. For example:

```
J: @M
```

causes the computer to skip over instructions until it reaches the next match statement and to begin execution at that point in the program.

Programmers can write a series of possible matches to one student response, with feedback messages or branching to different parts of the

program depending on which M: statement is the first to be successful. The use of the jump modifier allows this type of programming:

```
MJ:  RED
T:   I LIKE THAT!
J:   @P
M:   BLUE          (personalizing the lesson:  if the
                    word RED does not match, the program
                    jumps to the next match statement
                    and tries to match the student's
                    response with the word BLUE; if
                    RED does appear in the response,
                    the message is printed on the term-
                    inal and the computer jumps to the
                    next problem (PR:) statement)
```

Compute Command (C)

The op-code C: is used to assign initial values to variables so they may be used elsewhere, to carry out arithmetic calculations and to edit alphanumeric or string variables. Any alphanumeric constants (literals) used on a C: statement must be enclosed in quotation marks. Concatenation of two strings of alphanumeric variables can also be used on the C: statement. Functions can be referenced. Examples include:

```
C:  X=2            (assigns value of 2 to the nu-
                    meric variable, X)
C:  Y(1,2)=0       (assigns value of 0 to one
                    element of an array--the array
                    must be dimensioned before as-
                    signing values to the individual
                    variables)
C:  X$="YES"       (assigns value of YES to X$; var-
                    iable must be dimensioned)
C:  X=2*SIN(Z)     (assigns value to X--the variable
                    Z must be assigned a value before
                    using it on the right-hand side)
```

The C: statement can perform editing on alphanumeric variables. This is accomplished by naming the variable and listing the edit options which are wanted. More than one option may be included on one C: statement, but these options are executed from left to right in the order they appear. Example include:

```
C:  X$="Yes"      (assigns value Yes to X$)
C:   \X$\U        (converts all letters to capitals;
                   new value is YES)
C:  Y$="yes"      (assigns initial value to Y$)
C:   \Y$\C        (capitalizes only the first letter;
                   new value is Yes
C:   \X$\SP       (replaces each S found in X$ with
                   the letter P; new value is YEP)
C:   \Y$ \Y       (removes each Y found in Y$; new
                   value is es)
C:   \Y$ \S \U    (options can be combined:  this
                   statement removes each s found in
                   Y$ and capitalizes the remaining
                   letters in the string; new value
                   is E)
```

Note that every alphanumeric variable must be dimensioned before the programmer can use it in the C: statement or elsewhere. When more than one option is used in editing, a space must separate each option. For example: C: \Z$ \C \U removes @ characters from Z$ and capitalizes the letters left in the variable. A space must always follow the name of the variable.

Conditions can be placed on the C: so that only when the conditions are satisfied will the instruction be executed. This can be useful when a check for 0 in the denominator is needed. For example:

```
C(X < > 0):  Y=5/X        (will be skipped if X=) so
                          that no attempt is made to
                          divide unless the problem
                          can be worked)
```

Type Command (T)

The T: command is used to output messages to the student during the program. Variables may be placed on a T: statement and, if the variable has been assigned a value and is properly formatted, the value replaces the variable name in the printout. Numeric variables must be preceded by a # sign and alphanumeric variables must be preceded by

a $ sign. The name of the variable must be followed by a space. Array elements may not be placed on a T: statement. Examples include:

```
T:  This is a math lesson.          (causes the message to
                                    be printed)
C:  X=52.5                          (assigns value to X)
C:  A$="JOE"                        (assigns value to A$;
                                    note that A$ had to be
                                    dimensioned earlier in
                                    this program)
T:  WELL, $A$ ,YOU SCORED #X .      (causes the statement
                                    WELL, JOE, YOU SCORED
                                    52.5. to be printed)
```

Execution of a T: statement causes the terminal to output the message and advance the paper (or cursor on a CRT) to the start of the next line. If the programmer wishes to leave the paper or cursor at the end of the line, the H modifier can be used. For example:

```
T:  (spaces or just the return key causes one blank
    line to be printed and the paper advanced to the
    next line)
TH: WHAT IS YOUR NAME?              (prints the message,
                                    but leaves the paper
                                    or cursor hanging;
                                    when the student en-
                                    ters his/her name, it
                                    will appear on the
                                    terminal immediately
                                    following the question
                                    mark)
```

This feature can be used to allow students to fill in the blanks.

```
D:  B$(5)                           (reserves 5 memory
                                    locations for B$)
C:  B$=CHR(8)!!CHR(8)!!CHR(8)
    !!CHR(8)!!CHR(8)                (loads five back-
                                    space characters
                                    into B$)
T:  FILL IN THE BLANK:              (outputs the instruc-
                                    tions)
TH: THE FORMULA FOR SULFURIC
    ACID IS _____ $B$               (causes the message to
                                    be printed and the cur-
                                    sor to be positioned
                                    back at the start of
                                    the underline)
```

The E conditioner can be used on a T: statement to cause the message to be printed only in an error condition exists. The errors include (1) no numerals input when one is expected, (2) a mistake on a C: statement, or (3) an error in a relational expression in any statement. The error is cleared if the next statement capable of raising an error is executed properly. This feature can be used to check the student's response to be sure a numeral was entered. For example:

```
C:  X=0                          (assigns initial value
                                 to X)
A:  #X                           (inputs student response
                                 and stores the first
                                 numeral in X; see the
                                 section on A: below)
TE: PLEASE TYPE IN A NUMBER      (causes the message to
                                 be printed only when
                                 an error occurs, that
                                 is, when the student
                                 does not enter a num-
                                 ber)
```

Both the T: statement and the Jump Command (discussed below) are able to be conditioned so that they execute only when the conditions are met. These conditioners include Y or N, which depend on the result of the preceding M: statement, digits (any digit from 0-9) which depend on the value of the system variable %A, and relations to check the values of variables to determine whether or not the statement is executed. Some examples are shown below:

```
TY: GOOD WORK!                   (causes the message to
                                 print only if the last
                                 executed        M:
                                 statement was successful;
                                 if it was not successful,
                                 this statement is skipped)
TN: BETTER TRY THAT AGAIN!       (causes the message to
                                 print only if the preceding
                                 M: statement was not suc-
                                 cessful)
```

T2: LET'S TRY ANOTHER PROBLEM.	(causes the message to be printed only if the value of %A is 2; that is, the student had to input the answer twice)
TY3: THIRD TIME'S THE CHARM!	(causes the message to be printed only if the preceding M: statement was successful and the value of %A is 3)
TY(%A=3): THIRD TIME'S THE CHARM!	(same as above, only instead of a digit, the relational expression %A=3 is used)
T(X<0!X>=1): TYPE IN A NUMBER BETWEEN 0 AND 1.	(causes the message to be printed if either condition or both is met; that is, only when the number is not between 0 and 1, including 1)
TC: PLEASE TRY AGAIN.	(causes the message to be printed only if the relational expression on the previous statement was true: if no relational expression was used on the previous statement, execution depends on the truth of the last relational expression used as a conditioner)

If the message to be printed is longer than one line, the message can be continued on the next line. For example:

```
T(X<>5): YOU DID NOT GET THE RIGHT ANSWER. THE PROBLEM WILL BE
: PRINTED AGAIN AND YOU CAN TRY A DIFFERENT ANSWER.
```

will cause both lines to be printed if X is not equal to 5. If X=5 then both lines are skipped. The same conditions apply to the second line as were used in the original T: statement, so the programmer can save some typing by using the colon to continue the message instead of rewriting the conditions. Each line of continued text must contain at least one character to be printed.

Jump Command (J)

The op-code J: is used to cause changes in the normal sequential execution of Pilot commands by skipping or repeating parts of the program. When a J: statement is encountered, control is passed to the statement which is identified as the destination. For example:

```
J:  @M      (causes the computer to skip every
            instruction until the next M: state-
            ment is encountered)
J:  @P      (causes the computer to skip every
            instruction until the next PR:
            statement is encountered)
J:  @A      (causes the computer to backtrack
            until it finds the last A: state-
            ment which was executed and start
            again from that point--see the
            section on the Accept Command)
```

Labels can be used as the destination for a J: statement and the computer will search for the instruction which has that label and begin execution at that point. For example:

```
*PROB1                  (label for first problem)
T:  HOW MUCH IS 2+2?    (outputs first question)
.                       (more commands to input
.                       and judge the student re-
.                       sponse)
J:  PROB1               (causes the computer to
                        jump to the line which
                        has the label, PROB1)
```

The J: statement can also be conditioned using the relational expression, Y or N condition depending on the last executed M: statement, the E condition so that the jump occurs only if an error occurred, and digits which refer back to how many times the same A: statement has been executed. Examples of some of these conditions are:

```
JY: HELP                (causes the program to
                        jump to the instruction
                        with label HELP if the
                        preceding M: statement
                        was successful)
```

JN: @M	(causes the program to jump to the next M: statement if the preceding M: statement was unsuccessful)
JE: @A	(causes the program to jump to the last previous A: statement if the error condition is present--errors include no numerals when one is expected in the student response and incorrect expressions)
JY2(D < 3): PART7	(causes the program to jump to the instruction with label PART7 if all three conditions are met, that is, the last M: statement was successful, the system variable %A has a value of 2, and the variable D has a value less than 3--see the section on the Accept Command for more on the %A variable)

If the above conditions were not met, then the J: statement would be skipped and execution would continue sequentially.

Accept Command (A)

The op-code A: is used to input student responses. The response is edited according to whatever options are in effect (see the section on the Problem Command) and the answer is placed in the system variable %B. The other system variable %A is incremented each time the same A: statement is executed. It (%A) keeps track of the number of consecutive times the same A: statement is executed, but never has a value larger than 9. Examples:

A:	(inputs the student response, edits it, and stores it in %B)
A: #X	(inputs the student response, edits it, stores it in %B and places the first numerical value in the variable X--the number sign # must precede the variable name--if no numerals appear in the student response, the error condition is raised)

```
A:  $S$      (inputs the student response, edits it,
             stores it in %B, and also stores it in
             the variable S$--the dollar sign $ must
             appear before the variable name--vari-
             ables also must be used in a C: statement
             prior to use on the A: statement and
             dimensioned, if necessary)
AX:          (causes the student answer to be input
             exactly as entered with no editing what-
             soever and stored in the system variable
             %B)
A:  #X $Y$   (inputs the student response, edits it,
             stores it in %B and Y$, and stores the
             first numeral in the variable X)
```

The A: statement can also be used as the destination for a J: statement as discussed above by use of J: @A which causes the computer to find the last A: statement which was executed and start there again.

Use Command (U)

The op-code U: can be helpful in saving space in programs. When a U: statement is encountered, the computer jumps to the indicated destination and continues until an E: statement (not to be confused with the E conditioner) is found. The computer then jumps back to the statement which follows the U: instruction. For example:

```
U:  INPUT                          (causes the computer to
T:  OKAY, HERE GOES!               jump to the statement
.                                  with label INPUT--the
.                                  program then continues
.                                  until the computer en-
.                                  counters an E: statement
.                                  --at that time, the com-
.                                  puter jumps back up to
.                                  the command which follows
.                                  the U: statement--in
*INPUT                             this case, back to the
T:  DO YOU WANT TO CONTINUE?       T: OKAY . . . statement
  (YES OR NO)                      and execution continues
A:                                 normally from that point)
M:  YES
.
.
.
E:
```

The section of the program from the label to the E: statement is called a subroutine. Using subroutines can save space in the program if the same thing must be done several times. One U: statement can then be placed wherever you want to execute the subroutine. It is suggested that all subroutines be placed near the end of your program.

The destination of the U: statement does not have to be a label--you can use any of the destinations discussed above under the J: command, but to avoid problems, it is advised that you only use labels.

End Command (E)

The op-code E: is used to indicate the end of the program and the end of a subroutine. If an E: command is encountered at any other time, problems may cause the program to stop running. Normally the command appears as shown below:

```
E:
```

However, it is possible to declare a destination on the E: statement such as E: HELP which would cause the computer to jump to the instruction with label HELP. This is not recommended as it can cause problems that may stop the program or cause it to behave strangely!

Execute Indirect Command (XI)

The op-code XI: is used to allow the student to have more control over the program or to provide the program with more variety. When the XI: statement is encountered, the computer treats the rest of the statement, an alphanumeric variable, as if it were the command to be executed. For example:

```
D:  S$(9)              (dimensions alphanumeric variable,
                        S$)
C:  S$="T:  HELLO."    (sets the value of the variable
                        equal to the T: HELLO. statement)
```

```
XI:  S$            (causes the computer to execute
                    the T: HELLO. statement contained
                    in S$, so the computer types
                    HELLO. on the terminal)
```

To provide more variety, several alternative problems can be included in a lesson and the computer chooses randomly the one the student receives. If PROBØ, PROB1, PROB2, . . ., PROB9 are the names of 10 alternative problems:

```
D:  S$(50)          (dimensions the alphanumeric
                    variable, S$)
C:  S$="J:  PROB"   (this statement assigns the
  !!STR(INT(10*RND  value of "J:  PROB" concatenated
  (.25)))           with the numeral of the num-
                    ber between 0 and 9 inclusive
                    which is chosen by the computer
                    --see the section on Functions
                    for descriptions of STR, INT
                    and RND)
XI:  S$             (causes the computer to execute
                    the J: statement contained in
                    S$, which jumps to the label
                    of the problem which the com-
                    puter chose in the previous C:
                    statement)
```

Use of the XI: statement can also allow students to use the computer as a calculator if students understand the format used by the Pilot language for expressions.

```
D:  F$(72)          (dimensions alphanumeric variable
                    F$ to hold a maximum of 72 char-
                    acters)
T:  ENTER YOUR      (notifies student to use
  PROBLEM:          calculator)
A:  $F$             (inputs student problem and
                    stores it in F$--for example,
                    student could type 1000*2/SIN
                    (6.28)
C:  F$="C:  X="     (concatenates the value C:
  !!F$              X= with the student's problem)
XI:  F$             (causes the computer to execute
                    the C: statement contained in
                    F$ and compute the value of
                    X=1000*2/SIN(6.28)
T:  #X              (causes the computer to print
                    the value of X which was just
                    calculated to be printed on
                    the terminal)
```

GOTO Command (entered by student)

If the programmer tells the student the label names used in the program and allows the student to use the GOTO command (see the PR: statement options), any time the student is requested to input a response, he/she may type GOTO followed by the name of a label. This gives the student some control of the path used in completing the lesson. The computer accepts this GOTO and executes it just as it would a J: statement and the computer continues execution at the instruction which has that label.

This can also be a useful tool for finding "bugs" or mistakes in a Pilot program. The programmer can enter the GOTO command to see a section of the program over again to find errors. For example:

```
*START PR:  G        (label to start program and a prob-
                      lem statement to allow use of the
                      GOTO command)
T:  TEST2.
.
.
.
*PART1               (label for next section of program)
```

The programmer could enter GOTO START or GOTO PART1 any time the computer requests an input and the computer would jump to the section of the program chosen. You can also use GOTO @P, GOTO @M, and GOTO @A to cause the computer to jump to the next PR: statement in the program, or the next M: statement, or the last executed A: statement. If you have a copy of the program, this can speed up the process of locating errors.

Escape or @ Command (entered by student)

If the student is familiar with the Pilot language and the programmer allows the student to use the @ command (see the PR: statement options), the student may enter the @ command followed by an instruction in the Pilot

language which can be executed. Whenever a student response is input, if the first character is @, the computer executes a subroutine, written by the programmer, before continuing with the program. This subroutine must be labelled SYSX and can allow a help sequence to be printed or commands to be executed:

```
PR:  E                  (declares that the student can enter
                         the @ command)
A:
T:  INSTRUCTION.
.
.
.
*SYSX M:  TROU-         (subroutine to handle @ command--checks
  BLE!HELP               student response to see if keywords HELP
TY:  HELP SEC-          or TROUBLE are present--if so, the help
  TION                   sequence is printed and the program jumps
:  YOU HAVE 2           back to the T: INSTRUCTION.  statement
  OPTIONS.               following the A: command at which the
.                        student entered the @ command)
.
.
EY:
C:  B$=%B               (if the keywords are not present, the
C:  \B$  \@\U           computer assumes that a Pilot command
X1:  B$                 is being entered--the @ sign is stripped
E:  @A                  from the student response, it is capi-
                        talized, and the command is executed--
                        the program then returns to input the
                        next response)
```

In this case, the student could type @HELP and the computer would print a help sequence to explain the options of the program. If the student wanted to skip the help section and knew Pilot, he/she could type @J: @P which would cause the computer to execute the J: statement and jump to the next PR: statement in the program. This can also be helpful in tracing logic errors in programs which are not working. If the programmer wanted to check on the value of a variable, say X, the command @T: #X could be entered and the computer would type the value of X on the terminal. A glossary of terms abSD could be used in SYSX.

Entering Pilot Programs on the Horizon

Once you have written a Pilot program and want to enter it into the computer and run it, you need to use the Text Editor. Find the diskettes which contain the DOS system, the program EDIT, and PILOT, and locate a diskette with room to store a program. Turn on the computer and follow the instruction manual to boot-up the system.

Next, place the diskette with EDIT in the drive numbered 1 and the diskette you want to use for your program into drive 2. Dream up a name for your program such as MATH1 and enter the system commands:

```
CR MATH1,2 20
TY MATH1,2 Ø
GO EDIT
```

These commands create the file in which your program will be stored and load the Text Editor into the computer.

The program requests the names for INPUT and OUTPUT files. If you are just starting to write the program, press the return key after INPUT or @ followed by the return key and type the name of your file followed by a comma and the drive number 2 after OUTPUT. (To make changes or additions later, type the file name both times.)

To begin entering the Pilot program, type I followed by the first instruction line. At the end of the line, press the return key and begin typing the next line. Continue typing one command per line and pressing the return key at the end of each line. If you make a mistake, you can backspace and correct it, so it is good practice to check each line before you hit the return key. If you make a mistake and notice it after hitting the return key, it is more difficult to correct. After

you have finished entering the program, press the ESCAPE (ESC) key twice. Two dollar signs are printed. Then type E followed by two ESC keys. This will cause your program to be stored on the diskette.

Sample Run: (underlined letters are typed by the programmer)

```
INPUT? @
OUTPUT? MATH1,2
@IR:  THIS PROGRAM IS FOR MATH CLASS.
PR:  U  S  G  E
.
.
.
E:
$$                          (ESCAPE key)
@E$$
```

If you need to correct an error, type B followed by two ESCAPE keys. This moves you back to the start of the program. If you want to correct the Nth line, type NL followed by two ESCAPE keys. For example, to correct the 10th line, type 10L followed by two ESCAPE keys. This moves you to the start of the 10th line. To correct the mistake, type C followed by the incorrect text, then an ESCAPE key, followed by the correct text, then two ESCAPE keys. For example if the 12th line was entered incorrectly as:

```
T:  I AM THE COMPUTR.
```

Before dumping the program to the diskette, the programmer could correct the mistake by entering the following commands:

```
@B$$
@12L$$
@CCOMPUTR$COMPUTER$$
```

After all corrections are made, type the E$$ command to dump the program to diskette. (The L command moves you down however many lines you want, but if you type a negative number in front of the L, it moves you back up that many lines.) The programmer should refer to the manual on the

Text Editor for more information on how to use the commands. For example, if you are correcting a previously stored program, you must first use the A command to load the old program into memory before making the changes. Before typing I or C, type 20A$$ to read the program from diskette.

The D command is used to delete letters or words, M to move the pointer forward or backwards within a line, T to type or display lines on the terminal, and S to search for a group of characters somewhere in the program.

Running Pilot Programs

Once you have entered and stored a Pilot program on a diskette, place the diskette in the drive numbered 2. Place the diskette with the Pilot language in drive 1. Type GO PILOT followed by a return. This loads the Pilot language into the computer. The computer will request the program name. Type the name of the program you want to run followed by a comma and the drive number (2). For example, to run MATH1, the programmer or student would type:

```
GO PILOT
PROGRAM NAME? MATH1,2
```

The computer would then find the program MATH1 and begin execution.

APPENDIX A

To properly use vectors or arrays in Pilot, it is necessary to first initialize the values of all the variable locations. This can be done by using a C: statement for each possible location or setting up a loop to save room for larger arrays. For example:

```
D:  V(4)                 (sets up vector V with locations
                          for V(0), V(1), . . ., V(4)
C:  I=0                  (initializes variable I to be used
                          as an index)
*LOOP C:  V(I)=0         (sets the element of V indicated
                          by the index I to a value of 0)
C(I<4): I=I+1            (increments I if it is not already
                          to its maximum value of 4)
JC:  LOOP                (jumps back to the statement la-
                          belled LOOP if I is less than 4--
                          otherwise the program continues)
D:  A(5,5)               (dimensions the array A with
                          locations for A(0,0), A(0,1),
                          A(0,2), . . ., A(0,5), A(1,0),
                          A(1,1), . . ., A(1,5), A(2,0),
                          . . ., A(5,5)
C:  I=0                  (initializes variable I for use
                          as an index)
C:  I2=0                 (initializes variable I2 for use
                          as second subscript)
*LOOP1 C:  A(I,I2)=0     (sets the element of A indicated
                          by the indices I and I2 to a
                          value of 0)
C:  I2=I2+1              (increments index I2)
C(I2=6): I2=0            (resets I2 back to 0 if it has
                          passed the maximum value of 5)
CC:  I=I+1               (increments index I if I2 has
                          passed its maximum value)
J(I<6):  LOOP1           (jumps back to the statement
                          labelled LOOP1 if I has not
                          exceeded its maximum value--if
                          I is 6, continues with the pro-
                          gram)
```

APPENDIX B

Exponential Notation and ASCII Code

Pilot allows the user to specify numerical constants in either normal decimal notation (.0025, 10038) or in exponential notation (2.5E-3, 1.0038E4). The exponent is the power of 10 which must be multiplied by the number in front of the E in order to get the number which is represented. No spaces are allowed in exponential notation.

When using the CHR function or comparing alphanumeric variables, the ASCII codes represent the corresponding values of various characters. Some of the characters which are useful are shown below with their ASCII codes values; the first six characters may vary with the terminal used.

Value	Character	Value	Character	Value	Character
0	NUL	55	7	83	S
8	backspace	56	8	84	T
10	line feed	57	9	85	U
12	form feed	58	:	86	V
13	carriage return	59	;	87	W
32	blank or space	60	<	88	X
33	!	61	=	89	Y
34	"	62	>	90	Z
35	#	63	?	91	[
36	$	64	@	92	\
37	%	65	A	93	]
38	&	66	B	94	^ or ↑
39	'	67	C	95	← or ←
40	(	68	D	97	a
41	)	69	E	98	b
42	*	70	F	99	c
43	+	71	G	100	d
44	,	72	H	101	e
45	-	73	I	102	f
46	.	74	J	103	g
47	/	75	K	104	h
48	0	76	L	105	i
49	1	77	M	106	j
50	2	78	N	107	k
51	3	79	O	108	l
52	4	80	P	109	m
53	5	81	Q	110	n
54	6	82	R	111	o

Value	Character
112	p
113	q
114	r
115	s
116	t
117	u

Value	Character
118	v
119	w
120	x
121	y
122	z
126	~